

BOOK 4: FUNCTIONS

Chapter 1: What Are Functions

NAMING A PIECE OF CODE

You have been using functions without knowing it:

```
console.log("Hello")
```

The `console.log` is a function. Someone wrote code that displays things, gave it a name, and now you can use it by calling that name.

A function is a named block of code you can run whenever you want.

Consider this code:

```
console.log("=====")
console.log("WELCOME")
console.log("=====")
```

```
console.log("Processing...")
```

```
console.log("=====")
console.log("GOODBYE")
console.log("=====")
```

The separator pattern repeats. With a function, you write it once and use it anywhere:

```
function printSeparator() {
  console.log("=====")
}
```

```
printSeparator()
console.log("WELCOME")
printSeparator()
```

```
console.log("Processing...")
```

```
printSeparator()  
console.log("GOODBYE")  
printSeparator()
```

Same output, but the separator code exists in one place. Change it once, it changes everywhere.

CREATING A FUNCTION

The basic structure:

```
function functionName() {  
  code to run  
}
```

The function keyword starts the definition.

The name follows JavaScript naming rules (camelCase is standard).

Parentheses () come after the name.

Curly braces {} contain the code.

```
function sayHello() {  
  console.log("Hello!")  
}
```

This creates the function. It does not run it yet. The code inside is stored for later.

CALLING A FUNCTION

To run the function, call it by name with parentheses:

```
function sayHello() {
```

```
console.log("Hello!")  
}
```

```
sayHello()
```

Output: Hello!

The parentheses are required. They tell JavaScript to execute the function, not just reference it.

Without parentheses:

```
console.log(sayHello)
```

This prints something like: [Function: sayHello]

It shows the function itself, not its result. Parentheses execute it.

FUNCTIONS CAN BE CALLED MULTIPLE TIMES

Once defined, call a function as many times as you want:

```
function greet() {  
  console.log("Welcome!")  
  console.log("Have a nice day!")  
}
```

```
greet()  
greet()  
greet()
```

Output:

Welcome!

Have a nice day!

Welcome!

Have a nice day!

Welcome!

Have a nice day!

Three calls, three executions.

FUNCTIONS RUN WHERE CALLED

The function runs at the point where you call it:

```
console.log("1. Before function definition")
```

```
function showMessage() {  
  console.log("3. Inside function")  
}
```

```
console.log("2. After definition, before call")
```

```
showMessage()
```

```
console.log("4. After function call")
```

Output:

1. Before function definition
2. After definition, before call
3. Inside function
4. After function call

The function definition does not run the code. Only the call does.

WHY USE FUNCTIONS

Avoid repetition:

Without function:

```
console.log("Loading...")
let start = Date.now()
let loadTime = Date.now() - start
console.log(`Done in ${loadTime}ms`)
```

```
console.log("Loading...")
let start2 = Date.now()
let loadTime2 = Date.now() - start2
console.log(`Done in ${loadTime2}ms`)
```

```
console.log("Loading...")
let start3 = Date.now()
let loadTime3 = Date.now() - start3
console.log(`Done in ${loadTime3}ms`)
```

Same pattern three times. With a function, write once.

Organize code:

```
function processOrder() {
  validateCart()
  calculateTotal()
  applyDiscount()
  processPayment()
  sendConfirmation()
}
```

Functions break big tasks into named steps. Reading `processOrder` tells you what happens even before seeing the details.

Make changes in one place:

```
function formatPrice(price) {
  return "$" + price.toFixed(2)
}
```

If you later need to change to euros or add commas for thousands, change one function. All uses update automatically.

FUNCTION NAMES

Good function names describe what the function does:

Good names:

calculateTotal

sendEmail

validatePassword

formatDate

showError

Poor names:

doStuff

function1

temp

x

Use verbs for actions:

get, set, calculate, validate, format, display, create, remove, update, check

If you cannot name a function clearly, it might be doing too much.

FUNCTIONS INSIDE FUNCTIONS

A function can call other functions:

```
function greetUser() {  
  printHeader()  
  printWelcome()  
  printFooter()  
}
```

```
function printHeader() {
```

```

console.log("=====")
}

function printWelcome() {
  console.log("Welcome, User!")
}

function printFooter() {
  console.log("=====")
}

greetUser()

```

Output:

```

=====
Welcome, User!
=====

```

This builds complex behavior from simple pieces.

ORDER OF DEFINITIONS

In JavaScript, function definitions are "hoisted" to the top. You can call a function before its definition appears:

```

sayHello()

function sayHello() {
  console.log("Hello!")
}

```

This works because JavaScript moves the function definition up before running the code.

However, for clarity, most programmers define functions before calling them. It makes code easier to read top to bottom.

PRACTICAL EXAMPLES

Print a border:

```
function printBorder() {  
  console.log("+ ----- +")  
}
```

```
printBorder()  
console.log("| Hello World! |")  
printBorder()
```

Show loading state:

```
function showLoading() {  
  console.log("Loading...")  
  console.log("Please wait...")  
}
```

```
showLoading()
```

Log an action:

```
function logAction(action) {  
  let timestamp = new Date().toISOString()  
  console.log(`[${timestamp}] ${action}`)  
}
```

PRACTICE

Create a function that prints your name.

Create a function that prints three lines of equals signs.

Create a function that calls two other functions.

Create a function with a meaningful name for something you would do in a program.

WHAT YOU LEARNED

You learned that functions are named blocks of code.

You learned to define functions with the function keyword.

You learned to call functions with parentheses.

You learned that functions can be called multiple times.

You learned to name functions with descriptive verbs.

You learned that functions help avoid repetition.

The next chapter teaches you how to give functions information to work with.

BOOK 4: FUNCTIONS

Chapter 2: Parameters and Arguments

GIVING FUNCTIONS INFORMATION

The greet function from the last chapter always prints the same message. What if you want to greet different people?

```
function greet(name) {  
  console.log(`Hello, ${name}!`)  
}
```

```
greet("Alice")  
greet("Bob")  
greet("Charlie")
```

Output:

Hello, Alice!
Hello, Bob!
Hello, Charlie!

The name inside the parentheses is a parameter. It acts like a variable that receives a value when the function is called.

The values you pass ("Alice", "Bob", "Charlie") are arguments.

Parameter: The variable name in the function definition.

Argument: The actual value passed when calling the function.

SINGLE PARAMETER

```
function square(number) {  
  console.log(number * number)  
}
```

```
square(5)  
square(10)  
square(3)
```

Output:

```
25  
100  
9
```

The parameter number receives whatever value you pass. When you call square(5), number becomes 5 inside the function.

MULTIPLE PARAMETERS

Functions can have multiple parameters separated by commas:

```
function add(a, b) {  
  console.log(a + b)  
}
```

```
add(3, 4)  
add(10, 20)  
add(100, 200)
```

Output:

```
7  
30  
300
```

The first argument goes to the first parameter. The second argument goes to the second parameter.

```
function introduce(name, age, city) {  
  console.log(`My name is ${name}.`)  
  console.log(`I am ${age} years old.`)  
  console.log(`I live in ${city}.`)  
}
```

```
introduce("Alex", 25, "Seattle")
```

Output:

```
My name is Alex.  
I am 25 years old.  
I live in Seattle.
```

Order matters. If you swap arguments, the values go to wrong parameters.

PARAMETERS ARE LOCAL

Parameters exist only inside the function:

```
function greet(name) {
```

```
console.log(`Hello, ${name}`)  
}
```

```
greet("Alice")
```

```
console.log(name)
```

Error. name does not exist outside the function.

Each call gets its own copy:

```
function showNumber(num) {  
  console.log(`Number is: ${num}`)  
}
```

```
showNumber(5)  
showNumber(10)
```

The first call has num as 5. The second call has num as 10. They do not interfere.

PARAMETERS VS ARGUMENTS

These terms are often confused:

```
function greet(name) {  
  console.log(`Hello, ${name}`)  
}
```

```
greet("Alice")
```

name is the parameter (in the definition).
"Alice" is the argument (in the call).

Think of parameters as empty boxes waiting for values. Arguments are the values you put in the boxes.

In practice, people often use these terms interchangeably. The important thing is understanding that values flow from the call into the function.

DEFAULT PARAMETERS

You can give parameters default values:

```
function greet(name = "Guest") {  
  console.log(`Hello, ${name}!`)  
}
```

```
greet("Alice")  
greet()
```

Output:

Hello, Alice!

Hello, Guest!

If no argument is provided, the default value is used.

```
function createUser(name, role = "user", active = true) {  
  console.log(`Name: ${name}`)  
  console.log(`Role: ${role}`)  
  console.log(`Active: ${active}`)  
}
```

```
createUser("Alice")  
createUser("Bob", "admin")  
createUser("Charlie", "moderator", false)
```

Output:

Name: Alice

Role: user

Active: true

Name: Bob

Role: admin

Active: true

Name: Charlie

Role: moderator

Active: false

Default parameters must come after required parameters.

PASSING EXPRESSIONS

Arguments can be any expression:

```
function double(num) {  
  console.log(num * 2)  
}
```

```
double(5)  
double(3 + 4)  
double(10 / 2)
```

```
let x = 8  
double(x)  
double(x + 2)
```

Output:

```
10  
14  
10  
16  
20
```

The expression is evaluated first, then the result is passed.

```
function greet(name) {  
  console.log(`Hello, ${name}!`)  
}
```

```
let firstName = "John"
let lastName = "Doe"

greet(firstName + " " + lastName)
```

Output: Hello, John Doe!

PASSING VARIABLES

When you pass a variable, the function gets a copy of the value:

```
function addTen(num) {
  num = num + 10
  console.log(` Inside: ${num}`)
}

let myNumber = 5
console.log(` Before: ${myNumber}`)

addTen(myNumber)

console.log(` After: ${myNumber}`)
```

Output:
Before: 5
Inside: 15
After: 5

The original variable `myNumber` is unchanged. The function worked with a copy.

This is called "pass by value" for simple types like numbers, strings, and booleans.

PRACTICAL EXAMPLES

Calculate rectangle area:

```
function calculateArea(width, height) {  
  let area = width * height  
  console.log(`Area: ${area} square units`)  
}
```

```
calculateArea(5, 3)  
calculateArea(10, 10)  
calculateArea(7, 4)
```

Format currency:

```
function formatPrice(amount, currency = "$") {  
  console.log(`${currency}${amount.toFixed(2)}`)  
}
```

```
formatPrice(19.99)  
formatPrice(99, "EUR ")  
formatPrice(1234.5, "GBP ")
```

Create greeting based on time:

```
function greetByTime(name, hour) {  
  let greeting  
  
  if (hour < 12) {  
    greeting = "Good morning"  
  } else if (hour < 17) {  
    greeting = "Good afternoon"  
  } else {  
    greeting = "Good evening"  
  }  
  
  console.log(`${greeting}, ${name}!`)  
}
```

```
greetByTime("Alice", 9)
greetByTime("Bob", 14)
greetByTime("Charlie", 20)
```

Display user info:

```
function displayUser(username, email, isVerified = false) {
  console.log(`Username: ${username}`)
  console.log(`Email: ${email}`)
  console.log(`Verified: ${isVerified ? "Yes" : "No"}`)
  console.log("")
}
```

```
displayUser("alice_dev", "alice@example.com", true)
displayUser("new_user", "newbie@example.com")
```

REST PARAMETERS

When you do not know how many arguments will be passed, use rest parameters:

```
function sum(...numbers) {
  let total = 0
  for (let num of numbers) {
    total = total + num
  }
  console.log(`Sum: ${total}`)
}
```

```
sum(1, 2)
sum(1, 2, 3, 4, 5)
sum(10, 20, 30)
```

Output:

```
Sum: 3
Sum: 15
Sum: 60
```

The three dots (...) collect all arguments into an array.

You can combine regular parameters with rest:

```
function greetAll(greeting, ...names) {  
  for (let name of names) {  
    console.log(` ${greeting}, ${name}!`)  
  }  
}
```

```
greetAll("Hello", "Alice", "Bob", "Charlie")
```

Output:

Hello, Alice!

Hello, Bob!

Hello, Charlie!

The rest parameter must be last.

PRACTICE

Write a function that takes a name and age and prints a sentence about the person.

Write a function that takes two numbers and prints which one is larger.

Write a function with a default parameter for a greeting message.

Write a function that takes any number of numbers and prints the average.

WHAT YOU LEARNED

You learned that parameters receive values passed to functions.
You learned the difference between parameters and arguments.
You learned to use multiple parameters.
You learned that parameters are local to the function.
You learned to set default parameter values.
You learned about rest parameters for variable arguments.

The next chapter teaches you how functions can give back results.

BOOK 4: FUNCTIONS

Chapter 3: Return Values

GETTING RESULTS BACK

So far, our functions have printed output. But often you want a function to calculate something and give the result back:

```
function add(a, b) {  
  return a + b  
}
```

```
let sum = add(3, 4)  
console.log(sum)
```

Output: 7

The return statement sends a value back to where the function was called. The value replaces the function call.

HOW RETURN WORKS

```
function double(num) {  
  return num * 2  
}
```

```
}
```

```
let result = double(5)
```

Step by step:

1. double(5) is called
2. Inside the function, num becomes 5
3. return num * 2 calculates 10 and returns it
4. The call double(5) is replaced by 10
5. let result = 10 assigns the value

The function call becomes its return value.

USING RETURNED VALUES

Store in a variable:

```
let area = calculateArea(10, 5)  
console.log(area)
```

Use directly:

```
console.log(calculateArea(10, 5))
```

Use in expressions:

```
let totalArea = calculateArea(10, 5) + calculateArea(8, 6)
```

Use as arguments:

```
console.log(formatCurrency(calculateTotal(prices)))
```

Pass to conditions:

```
if (isValidPassword(password)) {  
  console.log("Access granted")  
}
```

```
}
```

Functions that return values are versatile. You choose what to do with the result.

RETURN ENDS THE FUNCTION

When return executes, the function stops immediately:

```
function checkAge(age) {  
  if (age < 0) {  
    return "Invalid age"  
  }  
  
  if (age < 18) {  
    return "Minor"  
  }  
  
  return "Adult"  
}
```

```
console.log(checkAge(-5))  
console.log(checkAge(12))  
console.log(checkAge(25))
```

Output:

Invalid age

Minor

Adult

Each return exits the function. Code after a return does not run:

```
function example() {  
  return 5  
  console.log("This never runs")  
}
```

The console.log after return is unreachable.

FUNCTIONS WITHOUT RETURN

If a function has no return statement, it returns undefined:

```
function greet(name) {  
  console.log(`Hello, ${name}`)  
}
```

```
let result = greet("Alice")  
console.log(result)
```

Output:
Hello, Alice
undefined

The function prints but returns nothing (undefined).

An empty return also gives undefined:

```
function doNothing() {  
  return  
}
```

```
console.log(doNothing())
```

Output: undefined

RETURNING DIFFERENT TYPES

Functions can return any type:

String:

```
function getGreeting(name) {  
  return `Hello, ${name}!`  
}
```

Number:

```
function calculateTax(amount) {  
  return amount * 0.08  
}
```

Boolean:

```
function isEven(num) {  
  return num % 2 === 0  
}
```

Array:

```
function getFirstThree(arr) {  
  return [arr[0], arr[1], arr[2]]  
}
```

Object:

```
function createUser(name, email) {  
  return {  
    name: name,  
    email: email,  
    createdAt: new Date()  
  }  
}
```

PRACTICAL EXAMPLES

Calculate area:

```
function rectangleArea(width, height) {  
  return width * height  
}
```

```
function circleArea(radius) {  
  return Math.PI * radius * radius  
}
```

```
let rect = rectangleArea(5, 3)  
let circ = circleArea(4)
```

```
console.log(`Rectangle: ${rect}`)  
console.log(`Circle: ${circ.toFixed(2)}`)
```

Validate input:

```
function isValidEmail(email) {  
  return email.includes("@") && email.includes(".")  
}
```

```
function isStrongPassword(password) {  
  return password.length >= 8  
}
```

```
if (isValidEmail("user@example.com")) {  
  console.log("Email is valid")  
}
```

Format data:

```
function formatPhoneNumber(number) {  
  let digits = number.replace(/\D/g, "")  
  return `(${digits.slice(0,3)}) ${digits.slice(3,6)}-${digits.slice(6)}`  
}
```

```
console.log(formatPhoneNumber("1234567890"))
```

Output: (123) 456-7890

Calculate discounts:

```
function calculateDiscount(price, percentage) {  
  let discount = price * (percentage / 100)  
  return price - discount  
}
```

```
let originalPrice = 100  
let salePrice = calculateDiscount(originalPrice, 20)
```

```
console.log(`Original: ${originalPrice}`)  
console.log(`Sale: ${salePrice}`)
```

Find maximum:

```
function max(a, b) {  
  if (a > b) {  
    return a  
  }  
  return b  
}
```

```
console.log(max(5, 3))  
console.log(max(10, 20))
```

Check ranges:

```
function isInRange(value, min, max) {  
  return value >= min && value <= max  
}
```

```
let temperature = 72
```

```
if (isInRange(temperature, 65, 78)) {  
  console.log("Comfortable temperature")  
}
```

EARLY RETURNS

Return early to handle special cases:

```
function divide(a, b) {  
  if (b === 0) {  
    return "Cannot divide by zero"  
  }  
  return a / b  
}
```

```
console.log(divide(10, 2))  
console.log(divide(10, 0))
```

Output:

5

Cannot divide by zero

```
function processUser(user) {  
  if (!user) {  
    return null  
  }  
  
  if (!user.isActive) {  
    return null  
  }  
  
  return user.name.toUpperCase()  
}
```

Early returns keep the main logic at the top level, avoiding deep nesting.

RETURNING VS PRINTING

Printing:

```
function addAndPrint(a, b) {
```

```
console.log(a + b)
}
```

```
addAndPrint(3, 4)
```

```
let result = addAndPrint(5, 6)
let doubled = result * 2
```

Output: 7, 11, then Error. result is undefined.

Returning:

```
function addAndReturn(a, b) {
  return a + b
}
```

```
let result = addAndReturn(5, 6)
let doubled = result * 2
```

```
console.log(doubled)
```

Output: 22

Functions that return values are more flexible. You can print, calculate, compare, or ignore the result. Functions that only print are limited.

Use return when you need the result. Use console.log for debugging or when display is the only purpose.

PRACTICE

Write a function that returns the larger of two numbers.

Write a function that returns true if a number is positive.

Write a function that returns a greeting string (not printing it).

Write a function that returns the sum of all numbers in an array.

Write a function that returns a formatted date string.

WHAT YOU LEARNED

You learned that `return` sends a value back from a function.

You learned that the function call is replaced by the return value.

You learned that `return` ends the function immediately.

You learned that functions without `return` give `undefined`.

You learned to use early returns for special cases.

You learned the difference between returning and printing.

The next chapter teaches you about function scope.

BOOK 4: FUNCTIONS

Chapter 4: Scope

WHERE VARIABLES EXIST

Scope determines where a variable can be accessed. Not all variables are available everywhere.

```
let outsideVariable = "I am outside"
```

```
function myFunction() {  
  let insideVariable = "I am inside"  
  console.log(outsideVariable)  
  console.log(insideVariable)  
}
```

```
myFunction()
```

```
console.log(outsideVariable)
console.log(insideVariable)
```

Output:

I am outside

I am inside

I am outside

Error: insideVariable is not defined

The variable inside the function cannot be accessed outside. The variable outside can be accessed anywhere.

LOCAL SCOPE

Variables declared inside a function are local to that function:

```
function sayHello() {
  let message = "Hello"
  console.log(message)
}
```

sayHello()

console.log(message)

Error. message exists only inside sayHello.

Each function has its own scope:

```
function functionA() {
  let x = 10
  console.log(`A: ${x}`)
}
```

```
function functionB() {
  let x = 20
  console.log(`B: ${x}`)
```

```
}
```

```
functionA()
```

```
functionB()
```

Output:

A: 10

B: 20

Both functions have a variable named x. They do not conflict because each x is local to its function.

GLOBAL SCOPE

Variables declared outside any function are global:

```
let globalVar = "I am global"
```

```
function accessGlobal() {  
  console.log(globalVar)  
}
```

```
accessGlobal()  
console.log(globalVar)
```

Output:

I am global

I am global

Global variables are accessible everywhere, inside and outside functions.

BLOCK SCOPE

Variables declared with `let` or `const` inside any block are scoped to that block:

```
if (true) {  
  let blockVar = "I am in a block"  
  console.log(blockVar)  
}
```

```
console.log(blockVar)
```

Error. `blockVar` exists only inside the `if` block.

This applies to loops too:

```
for (let i = 0; i < 3; i++) {  
  console.log(i)  
}
```

```
console.log(i)
```

Error. `i` exists only inside the loop.

SCOPE CHAIN

When you use a variable, JavaScript looks for it:

1. First in the current scope
2. Then in the outer scope
3. Then in the next outer scope
4. Until it reaches the global scope

```
let outerVar = "outer"
```

```
function outer() {  
  let middleVar = "middle"
```

```
  function inner() {
```

```
let innerVar = "inner"
console.log(innerVar)
console.log(middleVar)
console.log(outerVar)
}
```

```
inner()
}
```

```
outer()
```

Output:
inner
middle
outer

The inner function can see variables from all enclosing scopes.

SHADOWING

A local variable can have the same name as an outer variable:

```
let name = "Global"
```

```
function greet() {
  let name = "Local"
  console.log(name)
}
```

```
greet()
console.log(name)
```

Output:
Local
Global

Inside greet, the local name "shadows" the global one. The global

variable is unaffected.

Shadowing can cause confusion. Avoid using the same name for different purposes.

MODIFYING GLOBAL VARIABLES

Functions can modify global variables:

```
let counter = 0

function increment() {
  counter = counter + 1
}

console.log(counter)
increment()
increment()
increment()
console.log(counter)
```

Output:

```
0
3
```

This works but is generally bad practice. Functions that modify global state are harder to understand and test.

Better approach: pass values in, return values out.

```
function increment(value) {
  return value + 1
}

let counter = 0
counter = increment(counter)
counter = increment(counter)
```

```
counter = increment(counter)
console.log(counter)
```

Output: 3

The function is predictable. Given the same input, it returns the same output.

LET VS VAR

The var keyword (older JavaScript) has different scoping rules:

```
if (true) {
  var varVariable = "I use var"
  let letVariable = "I use let"
}
```

```
console.log(varVariable)
console.log(letVariable)
```

Output:

I use var

Error: letVariable is not defined

var is function-scoped, not block-scoped. It leaks out of blocks.

Always use let (or const) instead of var. Modern code should never use var.

CONST FOR CONSTANTS

const creates a variable that cannot be reassigned:

```
const PI = 3.14159
```

PI = 3.14

Error. Cannot assign to a constant.

Use const for values that should not change:

```
const MAX_USERS = 100
const API_URL = "https://api.example.com"
const TAX_RATE = 0.08
```

const is also block-scoped like let.

PARAMETERS ARE LOCAL

Function parameters are local to the function:

```
function greet(name) {
  console.log( Hello, ${name}`)
}
```

greet("Alice")

console.log(name)

Error. name is local to greet.

CLOSURES (PREVIEW)

Functions remember their scope even after the outer function returns:

```
function createCounter() {
  let count = 0
```

```
return function() {  
  count = count + 1  
  return count  
}  
}
```

```
let counter = createCounter()
```

```
console.log(counter())  
console.log(counter())  
console.log(counter())
```

Output:

```
1  
2  
3
```

The inner function "closes over" the count variable. This is called a closure. We will explore closures more in advanced topics.

PRACTICAL EXAMPLES

Avoiding global pollution:

```
let result = 0
```

```
function addToResult(num) {  
  result = result + num  
}
```

Bad. Uses global variable.

```
function calculateSum(numbers) {  
  let result = 0  
  for (let num of numbers) {  
    result = result + num
```

```
}  
return result  
}
```

```
let sum = calculateSum([1, 2, 3, 4, 5])
```

Good. Everything is local.

Module pattern (grouping related code):

```
const calculator = {  
  add: function(a, b) {  
    return a + b  
  },  
  subtract: function(a, b) {  
    return a - b  
  }  
}
```

```
console.log(calculator.add(5, 3))  
console.log(calculator.subtract(10, 4))
```

Configuration constants:

```
const CONFIG = {  
  MAX_RETRIES: 3,  
  TIMEOUT: 5000,  
  BASE_URL: "https://api.example.com"  
}
```

```
function fetchData() {  
  console.log(`Fetching from ${CONFIG.BASE_URL}`)  
  console.log(`Timeout: ${CONFIG.TIMEOUT}ms`)  
}
```

PRACTICE

Create a function with a local variable and try to access it outside.

Create two functions that each have a variable with the same name.

Create a global variable and modify it from inside a function.

Use `const` for a value that should not change.

WHAT YOU LEARNED

You learned that scope determines where variables are accessible.

You learned that local variables exist only inside their function.

You learned that global variables are accessible everywhere.

You learned about block scope with `let` and `const`.

You learned that JavaScript searches up the scope chain.

You learned that shadowing hides outer variables.

You learned to prefer `let` and `const` over `var`.

The next chapter teaches you about arrow functions.

BOOK 4: FUNCTIONS

Chapter 5: Arrow Functions

A SHORTER SYNTAX

Arrow functions provide a compact way to write functions:

Traditional function:

```
function add(a, b) {  
  return a + b  
}
```

Arrow function:

```
const add = (a, b) => {  
  return a + b  
}
```

Even shorter:

```
const add = (a, b) => a + b
```

All three do the same thing.

THE ARROW SYNTAX

The arrow `=>` replaces the function keyword:

```
const functionName = (parameters) => {  
  function body  
}
```

For single expressions, you can omit the braces and return:

```
const double = (num) => num * 2
```

For single parameters, you can omit the parentheses:

```
const double = num => num * 2
```

For no parameters, use empty parentheses:

```
const sayHello = () => console.log("Hello!")
```

CONVERTING TRADITIONAL TO ARROW

Step by step:

Start with traditional:

```
function greet(name) {  
  return `Hello, ${name}!`  
}
```

Remove function keyword, add arrow after parameters:

```
const greet = (name) => {  
  return `Hello, ${name}!`  
}
```

For single expression, remove braces and return:

```
const greet = (name) => `Hello, ${name}!`
```

For single parameter, remove parentheses:

```
const greet = name => `Hello, ${name}!`
```

ARROW FUNCTIONS WITH ARRAY METHODS

Arrow functions shine with array methods:

Traditional:

```
let numbers = [1, 2, 3, 4, 5]  
  
let doubled = numbers.map(function(num) {  
  return num * 2  
})
```

Arrow:

```
let doubled = numbers.map(num => num * 2)
```

Much cleaner.

More examples:

```
let evens = numbers.filter(num => num % 2 === 0)
```

```
let sum = numbers.reduce((acc, num) => acc + num, 0)
```

```
numbers.forEach(num => console.log(num))
```

```
let found = numbers.find(num => num > 3)
```

WHEN TO USE BRACES

Use braces when you have multiple statements:

```
const processUser = (user) => {  
  let greeting = `Hello, ${user.name}`  
  console.log(greeting)  
  return greeting.length  
}
```

With braces, you must use return explicitly.

Without braces, the expression is returned automatically:

```
const processUser = user => `Hello, ${user.name}`
```

If your arrow function is complex, use braces and explicit return.

RETURNING OBJECTS

To return an object literal without braces, wrap in parentheses:

Wrong:

```
const createUser = name => { name: name }
```

JavaScript thinks the braces are a function body, not an object.

Right:

```
const createUser = name => ({ name: name })
```

The parentheses tell JavaScript this is an object literal.

```
const createPoint = (x, y) => ({ x: x, y: y })
```

```
let point = createPoint(3, 4)  
console.log(point)
```

Output: { x: 3, y: 4 }

ARROW FUNCTIONS VS TRADITIONAL

Arrow functions are great for:

- Short, simple functions
- Callbacks in array methods
- Single-expression returns
- Inline function arguments

Traditional functions are better for:

- Named functions at the top level
- Functions that need their own this (covered in objects)
- Constructor functions
- Functions with many statements

Most modern JavaScript code uses arrow functions for callbacks and short functions.

PRACTICAL EXAMPLES

Math operations:

```
const add = (a, b) => a + b
const subtract = (a, b) => a - b
const multiply = (a, b) => a * b
const divide = (a, b) => a / b
```

```
console.log(add(5, 3))
console.log(multiply(4, 7))
```

String operations:

```
const toUpper = str => str.toUpperCase()
const toLower = str => str.toLowerCase()
const capitalize = str => str[0].toUpperCase() + str.slice(1)
```

```
console.log(capitalize("hello"))
```

Output: Hello

Array transformations:

```
const prices = [10, 20, 30, 40, 50]

const withTax = prices.map(price => price * 1.08)
const expensive = prices.filter(price => price > 25)
const total = prices.reduce((sum, price) => sum + price, 0)

console.log(withTax)
console.log(expensive)
console.log(total)
```

Validation functions:

```
const isPositive = num => num > 0
const isEven = num => num % 2 === 0
```

```
const isLongEnough = str => str.length >= 8
const hasAtSign = str => str.includes("@")

if (isLongEnough(password) && hasAtSign(email)) {
  console.log("Valid inputs")
}
```

Comparators for sorting:

```
const numbers = [3, 1, 4, 1, 5, 9, 2, 6]

const ascending = [...numbers].sort((a, b) => a - b)
const descending = [...numbers].sort((a, b) => b - a)

console.log(ascending)
console.log(descending)
```

Output:

```
[1, 1, 2, 3, 4, 5, 6, 9]
[9, 6, 5, 4, 3, 2, 1, 1]
```

STORING VS IMMEDIATELY USING

Store for reuse:

```
const double = num => num * 2

console.log(double(5))
console.log(double(10))
```

```
let results = [1, 2, 3].map(double)
```

Use immediately (inline):

```
let results = [1, 2, 3].map(num => num * 2)
```

Both are valid. Use named functions when you reuse them or when

the logic is complex enough to deserve a name.

CHAINING WITH ARROW FUNCTIONS

Arrow functions make chains readable:

```
const users = [  
  { name: "Alice", age: 25, active: true },  
  { name: "Bob", age: 17, active: true },  
  { name: "Charlie", age: 30, active: false },  
  { name: "Diana", age: 22, active: true }  
]
```

```
const result = users  
  .filter(user => user.active)  
  .filter(user => user.age >= 18)  
  .map(user => user.name.toUpperCase())  
  .sort()
```

```
console.log(result)
```

Output: ["ALICE", "DIANA"]

Each step is clear and concise.

ARROW FUNCTIONS IN OBJECT METHODS

Be careful with this in arrow functions (covered in objects chapter):

```
const person = {  
  name: "Alice",  
  greet: () => {  
    console.log(`Hello, I am ${this.name}`)  
  }  
}
```

```
}
```

```
person.greet()
```

This might not work as expected. Arrow functions do not have their own this. Use traditional functions for object methods.

PRACTICE

Convert a traditional function to an arrow function.

Write an arrow function that returns the square of a number.

Use arrow functions with map to double all numbers in an array.

Use arrow functions with filter to get strings longer than 5 characters.

Write an arrow function that takes no parameters and returns the current date.

WHAT YOU LEARNED

You learned that arrow functions use `=>` instead of `function`.

You learned the short syntax for single expressions.

You learned to omit parentheses for single parameters.

You learned to wrap object literals in parentheses.

You learned that arrow functions excel with array methods.

You learned when to use arrow functions versus traditional functions.

The next chapter teaches you about callbacks and higher-order functions.

BOOK 4: FUNCTIONS

Chapter 6: Higher-Order Functions

FUNCTIONS AS VALUES

In JavaScript, functions are values like numbers or strings. You can:

Store them in variables:

```
const greet = function(name) {  
  return `Hello, ${name}!`  
}
```

```
console.log(greet("Alice"))
```

Pass them as arguments:

```
function runTwice(func) {  
  func()  
  func()  
}
```

```
runTwice(function() {  
  console.log("Running!")  
})
```

Output:

Running!

Running!

Return them from functions:

```
function createMultiplier(factor) {  
  return function(number) {  
    return number * factor  
  }  
}
```

```
const double = createMultiplier(2)
const triple = createMultiplier(3)
```

```
console.log(double(5))
console.log(triple(5))
```

Output:

```
10
15
```

WHAT IS A HIGHER-ORDER FUNCTION

A higher-order function is a function that:

- Takes a function as an argument, OR
- Returns a function

You have been using higher-order functions:

```
[1, 2, 3].map(num => num * 2)
[1, 2, 3].filter(num => num > 1)
[1, 2, 3].forEach(num => console.log(num))
```

map, filter, and forEach are higher-order functions. They take a function as an argument.

CALLBACKS

A callback is a function passed to another function:

```
function processData(data, callback) {
  console.log("Processing data...")
  let result = data.toUpperCase()
  callback(result)
```

```
}  
  
processData("hello", function(result) {  
  console.log(`Result: ${result}`)  
})
```

Output:

Processing data...

Result: HELLO

The callback function is called after processing completes.

With arrow function:

```
processData("hello", result => console.log(`Result: ${result}`))
```

COMMON CALLBACK PATTERNS

Event handling:

```
button.addEventListener("click", function() {  
  console.log("Button clicked!")  
})
```

Timer:

```
setTimeout(function() {  
  console.log("3 seconds passed")  
}, 3000)
```

Array operations:

```
let numbers = [1, 2, 3, 4, 5]
```

```
numbers.forEach(function(num) {  
  console.log(num)  
})
```

```
let doubled = numbers.map(function(num) {  
  return num * 2  
})
```

CREATING YOUR OWN HIGHER-ORDER FUNCTIONS

Apply a function to each element:

```
function forEach(array, callback) {  
  for (let i = 0; i < array.length; i++) {  
    callback(array[i], i)  
  }  
}
```

```
forEach([10, 20, 30], function(value, index) {  
  console.log(`Index ${index}: ${value}`)  
})
```

Output:

Index 0: 10

Index 1: 20

Index 2: 30

Transform elements:

```
function map(array, transform) {  
  let result = []  
  for (let item of array) {  
    result.push(transform(item))  
  }  
  return result  
}
```

```
let doubled = map([1, 2, 3], num => num * 2)  
console.log(doubled)
```

Output: [2, 4, 6]

Filter elements:

```
function filter(array, test) {  
  let result = []  
  for (let item of array) {  
    if (test(item)) {  
      result.push(item)  
    }  
  }  
  return result  
}
```

```
let evens = filter([1, 2, 3, 4, 5], num => num % 2 === 0)  
console.log(evens)
```

Output: [2, 4]

FUNCTIONS THAT RETURN FUNCTIONS

Create specialized functions:

```
function createGreeter(greeting) {  
  return function(name) {  
    return `${greeting}, ${name}!`  
  }  
}
```

```
const sayHello = createGreeter("Hello")  
const sayHi = createGreeter("Hi")  
const sayGoodbye = createGreeter("Goodbye")
```

```
console.log(sayHello("Alice"))  
console.log(sayHi("Bob"))  
console.log(sayGoodbye("Charlie"))
```

Output:

Hello, Alice!

Hi, Bob!

Goodbye, Charlie!

Create multipliers:

```
function multiply(factor) {  
  return num => num * factor  
}
```

```
const double = multiply(2)  
const triple = multiply(3)  
const quadruple = multiply(4)
```

```
console.log(double(10))  
console.log(triple(10))  
console.log(quadruple(10))
```

Output:

20

30

40

Create validators:

```
function createValidator(minLength) {  
  return function(input) {  
    return input.length >= minLength  
  }  
}
```

```
const isValidPassword = createValidator(8)  
const isValidUsername = createValidator(3)
```

```
console.log(isValidPassword("abc"))  
console.log(isValidPassword("password123"))  
console.log(isValidUsername("ab"))  
console.log(isValidUsername("alice"))
```

Output:

false

true

false

true

PRACTICAL EXAMPLES

Repeat an action:

```
function repeat(times, action) {  
  for (let i = 0; i < times; i++) {  
    action(i)  
  }  
}
```

```
repeat(3, i => console.log(`Iteration ${i}`))
```

Output:

Iteration 0

Iteration 1

Iteration 2

Run until condition:

```
function runUntil(condition, action) {  
  let count = 0  
  while (!condition()) {  
    action(count)  
    count++  
    if (count > 1000) break  
  }  
}
```

```
let x = 0  
runUntil(  
  () => x >= 5,
```

```
(i) => {
  console.log(`Step ${i}: x = ${x}`)
  x++
}
)
```

Compose functions:

```
function compose(f, g) {
  return function(x) {
    return f(g(x))
  }
}
```

```
const addOne = x => x + 1
const double = x => x * 2
```

```
const doubleThenAdd = compose(addOne, double)
const addThenDouble = compose(double, addOne)
```

```
console.log(doubleThenAdd(5))
console.log(addThenDouble(5))
```

Output:

```
11
12
```

```
doubleThenAdd: 5 * 2 = 10, 10 + 1 = 11
addThenDouble: 5 + 1 = 6, 6 * 2 = 12
```

Memoization (caching results):

```
function memoize(fn) {
  let cache = {}

  return function(arg) {
    if (cache[arg] !== undefined) {
      console.log("Using cached result")
      return cache[arg]
    }
  }
}
```

```
console.log("Computing...")
let result = fn(arg)
cache[arg] = result
return result
}
}
```

```
const slowSquare = memoize(function(n) {
  return n * n
})
```

```
console.log(slowSquare(5))
console.log(slowSquare(5))
console.log(slowSquare(10))
console.log(slowSquare(10))
```

Output:

Computing...

25

Using cached result

25

Computing...

100

Using cached result

100

SORTING WITH CALLBACKS

The sort method takes a comparison function:

```
let numbers = [10, 5, 8, 1, 7]
```

```
numbers.sort((a, b) => a - b)
```

```
console.log(numbers)
```

Output: [1, 5, 7, 8, 10]

The callback tells sort how to compare:

- Return negative: a comes before b
- Return positive: b comes before a
- Return zero: keep original order

Sort objects:

```
let users = [  
  { name: "Charlie", age: 35 },  
  { name: "Alice", age: 25 },  
  { name: "Bob", age: 30 }  
]
```

```
users.sort((a, b) => a.age - b.age)
```

```
console.log(users.map(u => u.name))
```

Output: ["Alice", "Bob", "Charlie"]

PRACTICE

Write a higher-order function that takes an array and a callback, and returns true if the callback returns true for any element.

Write a function that returns a function adding a specific number.

Use callbacks to transform an array of strings to uppercase.

Create a function that logs how long another function takes to run.

WHAT YOU LEARNED

You learned that functions can be stored in variables.
You learned that functions can be passed as arguments (callbacks).
You learned that functions can return other functions.
You learned what makes a higher-order function.
You learned to create your own higher-order functions.
You learned practical patterns like composition and memoization.

The next chapter teaches you about pure functions and side effects.

BOOK 4: FUNCTIONS

Chapter 7: Pure Functions and Side Effects

WHAT IS A PURE FUNCTION

A pure function:

1. Always returns the same output for the same input
2. Has no side effects (does not change anything outside itself)

```
function add(a, b) {  
  return a + b  
}
```

This is pure. `add(2, 3)` always returns 5. It does not modify any external state.

```
function double(num) {  
  return num * 2  
}
```

Pure. `double(5)` always returns 10.

```
function greet(name) {  
  return `Hello, ${name}!`  
}
```

Pure. `greet("Alice")` always returns "Hello, Alice!".

WHAT ARE SIDE EFFECTS

A side effect is any change outside the function:

Modifying a global variable:

```
let counter = 0
```

```
function increment() {  
  counter = counter + 1  
}
```

```
increment()
```

Side effect: changes counter.

Logging to console:

```
function logMessage(message) {  
  console.log(message)  
}
```

Side effect: outputs to console.

Modifying an argument:

```
function addItem(array, item) {  
  array.push(item)  
}
```

```
let myArray = [1, 2, 3]  
addItem(myArray, 4)
```

Side effect: modifies the original array.

Modifying the DOM:

```
function updatePage() {  
  document.body.innerHTML = "Hello"  
}
```

Side effect: changes the web page.

Making network requests:

```
function fetchData() {  
  fetch("https://api.example.com/data")  
}
```

Side effect: sends a network request.

IMPURE FUNCTIONS

Functions with side effects or inconsistent returns are impure:

```
let multiplier = 2
```

```
function multiply(num) {  
  return num * multiplier  
}
```

```
console.log(multiply(5))
```

```
multiplier = 3
```

```
console.log(multiply(5))
```

Output:

10

15

Same input, different output. The function depends on external state.

```
function getRandomNumber() {  
  return Math.random()  
}
```

```
console.log(getRandomNumber())  
console.log(getRandomNumber())
```

Different output each time. Depends on external randomness.

```
function getCurrentTime() {  
  return new Date()  
}
```

Different output every time you call it.

WHY PURE FUNCTIONS MATTER

Predictable:

```
let result = add(2, 3)
```

You know result is 5. No need to check what else is happening.

Testable:

```
function testAdd() {  
  if (add(2, 3) !== 5) {  
    console.log("FAIL: add(2, 3)")  
  }  
  if (add(-1, 1) !== 0) {  
    console.log("FAIL: add(-1, 1)")  
  }  
  console.log("All tests passed")  
}
```

Easy to test. No setup, no cleanup.

Cacheable:

```
const cache = {}
```

```
function expensiveCalculation(n) {  
  if (cache[n] !== undefined) {  
    return cache[n]  
  }  
  let result = n * n * n  
  cache[n] = result  
  return result  
}
```

Pure functions with the same input give the same output. You can cache results safely.

Parallelizable:

Pure functions can run in parallel because they do not share state.

Easier to reason about:

```
function processOrder(order) {  
  let subtotal = calculateSubtotal(order.items)  
  let tax = calculateTax(subtotal)  
  let total = subtotal + tax  
  return { ...order, subtotal, tax, total }  
}
```

Each function does one thing. The flow is clear.

MAKING FUNCTIONS PURE

Instead of modifying external state, return new values:

Impure:

```
let total = 0
```

```
function addToTotal(amount) {  
  total = total + amount  
}
```

```
addToTotal(10)  
addToTotal(20)  
console.log(total)
```

Pure:

```
function addToTotal(currentTotal, amount) {  
  return currentTotal + amount  
}
```

```
let total = 0  
total = addToTotal(total, 10)  
total = addToTotal(total, 20)  
console.log(total)
```

Instead of modifying arrays, return new arrays:

Impure:

```
function addItem(array, item) {  
  array.push(item)  
}
```

Pure:

```
function addItem(array, item) {  
  return [...array, item]  
}
```

```
let original = [1, 2, 3]  
let updated = addItem(original, 4)
```

```
console.log(original)  
console.log(updated)
```

Output:

[1, 2, 3]

[1, 2, 3, 4]

The original is unchanged.

IMMUTABILITY

Pure functions work with immutable data. Instead of changing data, create new data:

Mutating (impure):

```
function updateUser(user, name) {  
  user.name = name  
}
```

```
let user = { name: "Alice" }  
updateUser(user, "Bob")
```

```
console.log(user)
```

Output: { name: "Bob" }

Original changed.

Immutable (pure):

```
function updateUser(user, name) {  
  return { ...user, name: name }  
}
```

```
let user = { name: "Alice" }  
let updatedUser = updateUser(user, "Bob")
```

```
console.log(user)
```

```
console.log(updatedUser)
```

Output:

```
{ name: "Alice" }
```

```
{ name: "Bob" }
```

Original unchanged. New object created.

PRACTICAL EXAMPLES

Transform data without mutation:

```
function capitalizeNames(users) {  
  return users.map(user => ({  
    ...user,  
    name: user.name.toUpperCase()  
  }))  
}
```

```
let users = [{ name: "alice" }, { name: "bob" }]  
let capitalized = capitalizeNames(users)
```

```
console.log(users)  
console.log(capitalized)
```

Calculate derived values:

```
function calculateOrderTotal(items, taxRate) {  
  let subtotal = items.reduce((sum, item) => sum + item.price, 0)  
  let tax = subtotal * taxRate  
  return {  
    subtotal: subtotal,  
    tax: tax,  
    total: subtotal + tax  
  }  
}
```

```
let items = [{ price: 10 }, { price: 20 }, { price: 30 }]
let order = calculateOrderTotal(items, 0.08)

console.log(order)
```

Filter and transform:

```
function getActiveUserNames(users) {
  return users
    .filter(user => user.active)
    .map(user => user.name)
}
```

```
let users = [
  { name: "Alice", active: true },
  { name: "Bob", active: false },
  { name: "Charlie", active: true }
]
```

```
console.log(getActiveUserNames(users))
```

Output: ["Alice", "Charlie"]

SIDE EFFECTS ARE NECESSARY

Programs need side effects to be useful:

- Display output (console.log, DOM updates)
- Save data (files, databases)
- Send network requests
- Get user input

The goal is not to eliminate side effects but to:

1. Isolate them
2. Keep most functions pure
3. Push side effects to the edges of your program

```
function calculateResult(data) {  
  return data.map(x => x * 2)  
}
```

```
function displayResult(result) {  
  console.log(result)  
}
```

```
let data = [1, 2, 3]  
let result = calculateResult(data)  
displayResult(result)
```

calculateResult is pure. displayResult has a side effect but is separated.

PRACTICE

Identify which functions are pure and which are impure.

Rewrite an impure function to be pure.

Create a function that returns a new array with an item added (without modifying the original).

Create a function that returns a new object with updated properties.

WHAT YOU LEARNED

You learned that pure functions always return the same output for the same input.

You learned that pure functions have no side effects.

You learned that side effects include modifying external state, logging, and I/O.

You learned why pure functions are easier to test and reason about.

You learned to create new data instead of mutating existing data.
You learned that side effects are necessary but should be isolated.

The next chapter brings everything together with practice programs.

BOOK 4: FUNCTIONS

Chapter 8: Practice Function Programs

CALCULATOR LIBRARY

Build a complete calculator with functions:

```
function add(a, b) {  
  return a + b  
}
```

```
function subtract(a, b) {  
  return a - b  
}
```

```
function multiply(a, b) {  
  return a * b  
}
```

```
function divide(a, b) {  
  if (b === 0) {  
    return "Error: Division by zero"  
  }  
  return a / b  
}
```

```
function power(base, exponent) {  
  return Math.pow(base, exponent)  
}
```

```
function squareRoot(num) {
```

```
if (num < 0) {  
  return "Error: Cannot take square root of negative"  
}  
return Math.sqrt(num)  
}
```

```
function percentage(value, percent) {  
  return (value * percent) / 100  
}
```

```
function round(num, decimals = 2) {  
  let factor = Math.pow(10, decimals)  
  return Math.round(num * factor) / factor  
}
```

```
console.log("=== CALCULATOR ===")  
console.log(`5 + 3 = ${add(5, 3)}`)  
console.log(`10 - 4 = ${subtract(10, 4)}`)  
console.log(`6 * 7 = ${multiply(6, 7)}`)  
console.log(`20 / 4 = ${divide(20, 4)}`)  
console.log(`2^8 = ${power(2, 8)}`)  
console.log(`sqrt(144) = ${squareRoot(144)}`)  
console.log(`20% of 150 = ${percentage(150, 20)}`)  
console.log(`3.14159 rounded = ${round(3.14159, 2)}`)
```

```
let result = add(multiply(3, 4), divide(20, 5))  
console.log(`${3 * 4} + (20 / 5) = ${result}`)
```

STRING UTILITIES

A collection of string manipulation functions:

```
function capitalize(str) {  
  if (!str) return ""  
  return str[0].toUpperCase() + str.slice(1).toLowerCase()  
}
```

```
function titleCase(str) {  
  return str  
    .split(" ")  
    .map(word => capitalize(word))  
    .join(" ")  
}
```

```
function reverse(str) {  
  return str.split("").reverse().join("")  
}
```

```
function countWords(str) {  
  if (!str.trim()) return 0  
  return str.trim().split(/\s+/).length  
}
```

```
function truncate(str, maxLength, suffix = "...") {  
  if (str.length <= maxLength) return str  
  return str.slice(0, maxLength - suffix.length) + suffix  
}
```

```
function repeat(str, times) {  
  let result = ""  
  for (let i = 0; i < times; i++) {  
    result += str  
  }  
  return result  
}
```

```
function padLeft(str, length, char = " ") {  
  while (str.length < length) {  
    str = char + str  
  }  
  return str  
}
```

```
function padRight(str, length, char = " ") {  
  while (str.length < length) {  
    str = str + char  
  }  
}
```

```
return str
}
```

```
function removeSpaces(str) {
  return str.replace(/\s+/g, "")
}
```

```
console.log("=== STRING UTILITIES ===")
console.log(`capitalize("hello"): ${capitalize("hello")}`)
console.log(`titleCase("the quick brown fox"): ${titleCase("the quick
brown fox")}`)
console.log(`reverse("hello"): ${reverse("hello")}`)
console.log(`countWords("one two three"): ${countWords("one two
three")}`)
console.log(`truncate("Hello World", 8): ${truncate("Hello World",
8)}`)
console.log(`repeat("ab", 3): ${repeat("ab", 3)}`)
console.log(`padLeft("5", 3, "0"): ${padLeft("5", 3, "0")}`)
```

ARRAY UTILITIES

Functions for working with arrays:

```
function first(arr) {
  return arr.length > 0 ? arr[0] : undefined
}
```

```
function last(arr) {
  return arr.length > 0 ? arr[arr.length - 1] : undefined
}
```

```
function sum(arr) {
  return arr.reduce((total, num) => total + num, 0)
}
```

```
function average(arr) {
  if (arr.length === 0) return 0
}
```

```
return sum(arr) / arr.length
}
```

```
function min(arr) {
  if (arr.length === 0) return undefined
  return Math.min(...arr)
}
```

```
function max(arr) {
  if (arr.length === 0) return undefined
  return Math.max(...arr)
}
```

```
function unique(arr) {
  return [...new Set(arr)]
}
```

```
function flatten(arr) {
  return arr.reduce((flat, item) => {
    if (Array.isArray(item)) {
      return flat.concat(flatten(item))
    }
    return flat.concat(item)
  }, [])
}
```

```
function chunk(arr, size) {
  let result = []
  for (let i = 0; i < arr.length; i += size) {
    result.push(arr.slice(i, i + size))
  }
  return result
}
```

```
function shuffle(arr) {
  let result = [...arr]
  for (let i = result.length - 1; i > 0; i--) {
    let j = Math.floor(Math.random() * (i + 1))
    let temp = result[i]
    result[i] = result[j]
```

```
result[j] = temp
}
return result
}
```

```
console.log(" == = ARRAY UTILITIES == =")
let numbers = [3, 1, 4, 1, 5, 9, 2, 6]
console.log(`Array: ${numbers}`)
console.log(`first: ${first(numbers)}`)
console.log(`last: ${last(numbers)}`)
console.log(`sum: ${sum(numbers)}`)
console.log(`average: ${average(numbers)}`)
console.log(`min: ${min(numbers)}`)
console.log(`max: ${max(numbers)}`)
console.log(`unique: ${unique(numbers)}`)
console.log(`chunk(3): ${JSON.stringify(chunk([1,2,3,4,5,6,7], 3))}`)
`)
```

VALIDATION LIBRARY

Functions for validating user input:

```
function isRequired(value) {
  return value !== null && value !== undefined && value !== ""
}
```

```
function isEmail(email) {
  return email.includes("@") && email.includes(".")
}
```

```
function isMinLength(str, min) {
  return str.length >= min
}
```

```
function isMaxLength(str, max) {
  return str.length <= max
}
```

```
function isNumber(value) {  
  return !isNaN(parseFloat(value)) && isFinite(value)  
}
```

```
function isInteger(value) {  
  return Number.isInteger(Number(value))  
}
```

```
function isInRange(num, min, max) {  
  return num >= min && num <= max  
}
```

```
function isAlphanumeric(str) {  
  return /^[a-zA-Z0-9]+$/ .test(str)  
}
```

```
function isURL(str) {  
  return str.startsWith("http://") || str.startsWith("https://")  
}
```

```
function validatePassword(password) {  
  let errors = []
```

```
  if (!isMinLength(password, 8)) {  
    errors.push("Must be at least 8 characters")  
  }
```

```
  if (password === password.toLowerCase()) {  
    errors.push("Must contain uppercase letter")  
  }
```

```
  if (password === password.toUpperCase()) {  
    errors.push("Must contain lowercase letter")  
  }
```

```
  if (!/^\d/.test(password)) {  
    errors.push("Must contain a number")  
  }
```

```
return {  
  valid: errors.length === 0,  
  errors: errors  
}  
}
```

```
console.log("=== VALIDATION ===")  
console.log(`isEmail("test@example.com"):  
${isEmail("test@example.com")}`)  
console.log(`isEmail("invalid"): ${isEmail("invalid")}`)  
console.log(`isNumber("42"): ${isNumber("42")}`)  
console.log(`isNumber("abc"): ${isNumber("abc")}`)
```

```
let result = validatePassword("weak")  
console.log(`validatePassword("weak"): ${JSON.stringify(result)}`)
```

```
result = validatePassword("StrongPass123")  
console.log(`validatePassword("StrongPass123"):  
${JSON.stringify(result)}`)
```

FUNCTIONAL PROGRAMMING UTILITIES

Higher-order functions for data processing:

```
function compose(...functions) {  
  return function(value) {  
    return functions.reduceRight((acc, fn) => fn(acc), value)  
  }  
}
```

```
function pipe(...functions) {  
  return function(value) {  
    return functions.reduce((acc, fn) => fn(acc), value)  
  }  
}
```

```
function curry(fn) {
```

```
return function curried(...args) {  
  if (args.length >= fn.length) {  
    return fn(...args)  
  }  
  return function(...moreArgs) {  
    return curried(...args, ...moreArgs)  
  }  
}  
}
```

```
function debounce(fn, delay) {  
  let timeoutId  
  return function(...args) {  
    clearTimeout(timeoutId)  
    timeoutId = setTimeout(() => fn(...args), delay)  
  }  
}
```

```
function once(fn) {  
  let called = false  
  let result  
  return function(...args) {  
    if (!called) {  
      called = true  
      result = fn(...args)  
    }  
    return result  
  }  
}
```

```
function memoize(fn) {  
  let cache = {}  
  return function(arg) {  
    if (cache[arg] === undefined) {  
      cache[arg] = fn(arg)  
    }  
    return cache[arg]  
  }  
}
```

```
const addOne = x => x + 1
const double = x => x * 2
const square = x => x * x
```

```
const pipeline = pipe(addOne, double, square)
console.log("=== FUNCTIONAL UTILITIES ===")
console.log(`pipe(addOne, double, square)(3): ${pipeline(3)}`)
```

```
const add = curry((a, b, c) => a + b + c)
const addFive = add(5)
const addFiveAndTen = addFive(10)
console.log(`curry add(5)(10)(20): ${addFiveAndTen(20)}`)
```

```
const factorial = memoize(function(n) {
  console.log(`Computing factorial(${n})`)
  if (n <= 1) return 1
  return n * factorial(n - 1)
})
```

```
console.log(`factorial(5): ${factorial(5)}`)
console.log(`factorial(5) again: ${factorial(5)}`)
```

DATE AND TIME UTILITIES

Functions for working with dates:

```
function formatDate(date, format = "YYYY-MM-DD") {
  let year = date.getFullYear()
  let month = String(date.getMonth() + 1).padStart(2, "0")
  let day = String(date.getDate()).padStart(2, "0")

  return format
    .replace("YYYY", year)
    .replace("MM", month)
    .replace("DD", day)
}
```

```
function addDays(date, days) {  
  let result = new Date(date)  
  result.setDate(result.getDate() + days)  
  return result  
}
```

```
function daysBetween(date1, date2) {  
  let diff = Math.abs(date2 - date1)  
  return Math.floor(diff / (1000 * 60 * 60 * 24))  
}
```

```
function isWeekend(date) {  
  let day = date.getDay()  
  return day === 0 || day === 6  
}
```

```
function getWeekday(date) {  
  let days = ["Sunday", "Monday", "Tuesday", "Wednesday",  
    "Thursday", "Friday", "Saturday"]  
  return days[date.getDay()]  
}
```

```
function startOfDay(date) {  
  let result = new Date(date)  
  result.setHours(0, 0, 0, 0)  
  return result  
}
```

```
function endOfDay(date) {  
  let result = new Date(date)  
  result.setHours(23, 59, 59, 999)  
  return result  
}
```

```
console.log("=== DATE UTILITIES ===")  
let today = new Date()  
console.log(`formatDate(today): ${formatDate(today)}`)  
console.log(`addDays(today, 7): ${formatDate(addDays(today, 7))}`)  
console.log(`isWeekend(today): ${isWeekend(today)}`)
```

```
console.log(`getWeekday(today): ${getWeekday(today)}`)  
  
let date1 = new Date("2024-01-01")  
let date2 = new Date("2024-12-31")  
console.log(`daysBetween: ${daysBetween(date1, date2)}`)
```

WHAT YOU BUILT

You built six utility libraries:

- Calculator with math operations
- String manipulation tools
- Array utilities for data processing
- Input validation functions
- Functional programming helpers
- Date and time utilities

Each library demonstrates:

- Pure functions that return values
- Functions that take functions as arguments
- Functions that return functions
- Default parameters and error handling
- Real-world practical use cases

WHAT COMES NEXT

You now have powerful tools:

- Variables store data (Book 1)
- Decisions control flow (Book 2)
- Loops process collections (Book 3)
- Functions organize and reuse code (Book 4)

But you have been working mainly with individual values. Real programs work with collections of data: lists of users, arrays of products, sets of configuration options.

Book 5 teaches you arrays in depth. You will learn to:

- Store collections of related data
- Access and modify elements
- Use powerful array methods
- Work with multi-dimensional data
- Choose the right data structure

Arrays are one of the most important data structures in programming. Combined with functions, they let you process any amount of data efficiently.

You finished Book 4. You can write modular, reusable code. Now we tackle data structures.